PROFESSOR: Continuing in the theme of sorting in general, but in particular, binary search trees, which are a kind of way of doing dynamic sorting, if you will, where the elements are coming and going. And at all times, you want to know the sorted order of your elements by storing them in a nice binary search tree.

Remember, in general, a binary search tree is a tree. It's binary, and it has the search property. Those three things. This is a rooted binary tree. It has a root. It's binary, so there's a left child and a right child. Some nodes lack a right or left child. Some nodes lack both.

Every node has a key. This is the search part. You store key in every node, and you have this BST property, or also called the search property, that every node-- if you have a node the stores key x, everybody in the left subtree stores a key that's less than or equal to x, and everyone that's in the right subtree stores a key that's greater than or equal to x. So not just the left and right children, but every descendant way down there is smaller than x. Every descendent way down there is greater than x.

So when you have a binary search tree like this, if you want to know the sorted order, you do what's called an in-order traversal. You look at a node. You recursively visit the left child. Then you print out the root. Then you recursively visit the right child.

So in this case, we'd go left, left, print 11. Print 20. Go right. Go left. Print 26. Print 29. Go up. Print 41. Go right. Print 50. Print 65. Then check that's in sorted order. If you're not familiar with in-order traversal, look at the textbook. It's a very simple operation. I'm not going to talk about it more here, except we're going to use it.

All right, we'll get to the topic of today's lecture in a moment, which is balance. What we saw in last lecture and recitation is that these basic binary search trees, where when you insert a node you just walk down the tree to find where that item fits-- like if you're trying to insert 30, you go left here, go right here, go right here, and say, oh 30 fits here. Let's put 30 there.

If you keep doing that, you can do insert. You can do delete. You can do these kinds of searches, which we saw, finding the next larger element or finding the next smaller element, also known as successor and predecessor. These are actually the typical names for those operations.

You can solve them in order h time. Anyone remember what h was? The height. Yeah, good. The height of the tree. So h is the height of the BST. What is the height of the tree?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Sorry?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Log n? Log n would be great, but not always. So this is the issue of being balance. So in an ideal world, your tree's going to look something like this. I've drawn this picture probably the most in my academic career. This is a nice, perfectly balanced binary search tree. The height is log n. This would be the balance case. I mean, roughly log n. Let's just put theta to be approximate.

But as we saw at the end of last class, you can have a very unbalanced tree, which is just a path. And there the height is n. What's the definition of height? That's actually what I was looking for. Should be 6.042 material. Yeah?

AUDIENCE: Is it the length of the longest path always going down?

PROFESSOR: Yeah, length of the longest path always going down. So length of the longest path from the root to some leaf. That's right. OK, so this is--

I highlight this because we're going to be working a lot with height today. All that's happening here, all of the paths are length log n. Here, there is a path of length n. Some of them are shorter, but in fact, the average path is n over 2. It's really bad. So this is very unbalanced. I'll put "very." It's not a very formal term, but that's like the worst case for BSTs.

This is good. This does have a formal definition. We call a tree balanced if the height is order log n. So you're storing n keys. If your height is always order log n, we get a constant factor here. Here, it's basically exactly log n, 1 times log n.

It's always going to be at least log n, because if you're storing n things in a binary tree, you need to have height at least log n. So in fact, it will be theta log n if your tree is balanced.

And today's goal is to always maintain that your trees are balanced. And we're going to do that using the structure called AVL trees, which I'll define in a moment. They're the original way people found to keep trees balanced back in the '60s, but they're still kind of the simplest.

There are lots of ways to keep a tree balanced, so I'll mention some other balance trees later on. In particular, your textbook covers two other ways to do it. It does not cover AVL trees, so pay attention.

One more thing I wanted to define. We talked about the height of the tree, but I'd also like to talk about the height of a node in a tree. Can anyone define this for me? Yeah?

AUDIENCE: It's the level that the node is at.

PROFESSOR: The level that the node is at. That is roughly right. I mean, that is right. It's all about, what is the level of a node?

AUDIENCE: Like how many levels of children it has.

PROFESSOR: How many levels of children it has. That's basically right, yeah.

AUDIENCE: The distance from it to the root.

PROFESSOR: Distance from it to the root. That would be the depth. So depth is counting from above. Height is--

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yes, longest path from that node to the leaf. Note that's why I wrote this definition actually, to give you a hint.

Here I should probably say down to be precise. You're not allowed to go up in these paths.

[INAUDIBLE]. All right. Sorry. I've got to learn how to throw.

All right. So for example, over here I'm going to write depths in red. If you're taking notes it's OK. Don't worry. So length off the longest path from it down to a leaf. Well, this is a leaf, so its height is 0. OK. Yeah, I'll just leave it at that. It takes 0 steps to get from a leaf to a leaf.

This guy's not a leaf. It has a child, but it has a path of length one to a leaf. So it's one. This guy has a choice. You could go left and you get a path of length 1, or you could go right and get a path of length 2. We take the max, so this guy has height 2. This node has height 1. This node has height 3.

How do you compute the height of a node? Anyone? Yeah.

AUDIENCE: Max of the height of the children plus 1.

PROFESSOR: Right. You take the max of the height of the children. Here, 2 and 1. Max is 2. Add 1. You get 3. So it's going to always be-- this is just a formula. The height of the left child maxed with the height of the right child plus 1.

This is obviously useful for computing. And in particular, in lecture and recitation last time, we saw how to maintain the size of every tree using data structure augmentation. Data structure augmentation. And then we started with a regular vanilla binary search tree, and then we maintained-- every time we did an operation on the tree, we also updated the size of the subtree rooted at that node, the size field.

Here, I want to store a height field, and because I have this nice local rule that tells me how to compute the height of a node using just local information-- the height of its left child, the height of its right child. Do a constant amount of work here.

There's a general theorem. Whenever you have a nice local formula like this for updating your information in terms of your children, then you can maintain it using constant overhead. So we can store the height of every node for free. Why do I care? Because AVL trees are going to use the heights of the nodes.

Our goal is to keep the heights small. We don't want this. We want this. So a natural thing to do is store the heights. When they get too big, fix it. So that's what we're going to do.

Maybe one more thing to mention over here for convenience. Leaves, for example, have children that are-- I mean, they have null pointers to their left and right children. You could draw them explicitly like this. Also some nodes just lack a single child. I'm going to define the depths of these things to be negative 1. This will be convenient later on.

Why negative 1? Because then this formula works. You can just think about it. Like leaves, for example, have two children, which are negative 1. You take the max. You add 1. You get 0. So that just makes things work out.

We don't normally draw these in the pictures, but it's convenient that I don't have to do special cases when the left child doesn't exist and the right child doesn't exist. You could either do special cases or you could make this definition. Up to you.

OK. AVL trees. So the idea with an AVL tree is the following.

We'd like to keep the height order log n. It's a little harder to think about keeping the height order log n than it is to think about keeping the tree balance, meaning the left and right sides are more or less equal. In this case, we're going to think about them as being more or less equal in height.

You could also think about them being more or less equal in subtree size. That would also work. It's a different balanced search tree. Height is kind of the easiest thing to work with.

So if we have a node, it has a left subtree. It has a right subtree, which we traditionally draw as triangles. This subtree has a height. We'll call it HL for left. By the height of the subtree, I mean the height of its root.

And the right subtree has some height, r. I've drawn them as the same, but in general they might be different. And what we would like is that h sub l and h sub r are more or less the same. They differ by at most an additive 1.

So if I look at h sub l minus h sub r in absolute value, this is at most 1, for every node. So I have some node x. For every node x, I want the left and right subtrees to be almost balanced.

Now, I could say differ by at most 0, that the left and right have exactly the same heights. That's difficult, because that really forces you to have exactly the perfect tree. And in fact, it's not even possible for odd n or even n or something. Because at the very end you're going to have one missing child, and then you're unbalanced there.

So 0's just not possible to maintain, but 1 is almost as good, hopefully. We're going to prove that in a second. And it turns out to be easy to maintain in log n time.

So let's prove some stuff. So first claim is that AVL trees are balanced. Balanced, remember, means that the height of them is always order log n. So we're just going to assume for now that we can somehow achieve this property. We want to prove that it implies that the height is at most some constant times log n. We know it's at least log n, but also like it to be not much bigger.

So what do you think is the worst case? Say I have n nodes. How could I make the tree as high as possible? Or conversely, if I have a particular height, how could I make it have as few nodes as possible? That'd be like the sparsest, the least balanced situation for AVL trees. Yeah?

AUDIENCE: You could have one node on the last level.

PROFESSOR: One node on the last level, yeah, in particular. Little more. What do the other levels look like? That is correct, but I want to know the whole tree. It's hard to explain the tree, but you can explain the core property of the tree. Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: For every node, let's make the right side have a height of one larger than the left side. I think that's worth a cushion. See if I can throw better. Good catch. Better than hitting your eye.

So I'm going to not prove this formally, but I think if you stare at this long enough it's pretty obvious. Worst case is when-- there are multiple worst cases, because right and left are symmetric. We don't really care. But let's say that the right subtree has height one more than the left for every node.

OK, this is a little tricky to draw. Not even sure I want to try to draw it. But you basically draw it recursively. So, OK, somehow I've figured out this where the height difference here is 1. Then I take two copies of it. It's like a fractal. You should know all about fractals by now. Problem set two.

And then you just-- well, that's not quite right. In fact, I need to somehow make this one a little bit taller and then glue these together. Little tricky. Let's not even try to draw the tree. Let's just imagine this is possible. It is possible.

And instead, I'm going to use mathematics to understand how high that tree is. Or actually, it's a little easier to think about-- let me get this right. It's so easy that I have to look at my notes to remember what to write. Really, no problem.

All right, so I'm going to define n sub h is the minimum number of nodes that's possible in an AVL tree of height h. This is sort of the inverse of what we care about, but if we can solve the inverse, we can solve the thing.

What we really care about is, for n nodes, how large can the height be? We want to prove that's order log n. But it'll be a lot easier to think about the reverse, which is, if I fix the height to be h, what's the fewest nodes that I can pack in? Because for the very unbalanced tree, I have a height of n, and I only need to put n nodes. That would be really bad.

What I prefer is a situation like this, where with height h, I have to put in 2 to the h nodes. That would be perfect balance. Any constant to the h will do. So when you take the inverse, you get a log. OK, we'll get to that in a moment.

How should we analyze n sub h? I hear something. Yeah?

AUDIENCE: [INAUDIBLE] 2 to the h minus 1 [INAUDIBLE].

PROFESSOR: Maybe, but I don't think that will quite work out. Any-- yeah?

AUDIENCE: So you have only 1 node in the last level, so it would be 1/2 to the h plus 1.

PROFESSOR: That turns out to be approximately correct, but I don't know where you got 1/2 to the h plus 1. It's not exactly correct. I'll tell you that, so that your analysis isn't right.

It's a lot easier. You guys are worried about the last level and actually what the tree looks like, but in fact, all you need is this. All you need is love, yeah.

AUDIENCE: [INAUDIBLE].

PROFESSOR: No, it's not a half. It's a different constant. Yeah?

AUDIENCE: Start with base cases and write a recursive formula.

PROFESSOR: Ah, recursive formula. Good. You said start with base cases. I always forget that part, so it's good that you remember. You should start with the base case, but I'm not going to worry about the base case because it won't matter. Because I know the base case is always going to be n order 1 is order 1. So for algorithms, that's usually all you need for base case, but it's good that you think about it.

What I was looking for is recursive formula, aka, recurrence. So can someone tell me-- maybe even you-- could tell me a recurrence for n sub h, in terms of n sub smaller h? Yeah?

AUDIENCE: 1 plus [INAUDIBLE].

PROFESSOR: 1 plus n sub h minus 1. Not quite. Yeah?

AUDIENCE: N sub h minus 1 plus n sub h minus 2.

PROFESSOR: N plus-- do you want the 1 plus?

AUDIENCE: I don't think so.

PROFESSOR: You do. It's a collaboration. To combine your two answers, this should be the correct formula. Let me double check. Yes, whew. Good.

OK, why? Because the one thing we know is that our tree looks like this. The total height here is h. That's what we're trying to figure out.

How many nodes are in this tree of height h? Well, the height is the max of the two directions. So that means that the larger has height h minus 1, because the longest path to a leaf is going to be down this way.

What's the height of this? Well, it's one less than the height of this. So it's going to be h minus 2. This is where the n sub h minus 1 plus n sub h minus 2 come in.

But there's also this node. It doesn't actually make a big difference in this recurrence. This is the exponential part. This is like itty bitty thing. But it matters for the base case is pretty much where it matters. Back to your base case.

There's one guy here, plus all the nodes on the left, plus all the nodes on the right. And for whatever reason, I put the left over here and the right over here. And of course, you could reverse this picture. It doesn't really matter. You get the same formula. That's the point.

So this is the recurrence. Now we need to solve it. What we would like is for it to be exponential, because that means there's a lot of nodes in a height h AVL tree. So any suggestions on how we could figure out this recurrence? Does it look like anything you've seen before?

AUDIENCE: Fibonacci.

PROFESSOR: Fibonacci. It's almost Fibonacci. If I hid this plus 1, which you wanted to do, then it would be exactly Fibonacci. Well, that's actually good, because in particular, n sub h is bigger than Fibonacci. If you add one at every single level, the certainly you get something bigger than the base Fibonacci sequence.

Now, hopefully you know Fibonacci is exponential. I have an exact formula. If you take the golden ratio to the power h, divide by root 5, and round to the nearest integer, you get exactly the Fibonacci number.

Crazy stuff. We don't need to know why that's true. Just take it as fact. And conveniently phi is bigger than 1. You don't need to remember what phi is, except it is bigger than 1. And so this is an exponential bound. This is good news.

So I'll tell you it's about 1.618. And so we get is that-- if we invert this, this says n sub h is bigger than some phi to the h. This is our n, basically.

What we really want to know is how h relates to n, which is just inverting this formula. So we have, on the other hand, the phi to the h divided by root 5 is less than n.

So I got a log base phi on both sides. Seems like a good thing to do. This is actually quite annoying. I've got h minus a tiny little thing. It's less than log base phi of n. And I will tell you that is about 1.440 times log base 2 of n, because after all, log base 2 is what computer scientists care about.

So just to put it into perspective. We want it to be theta log base 2 of n. And here's the bound. The height is always less than 1.44 times log n.

All we care about is some constant, but this is a pretty good constant. We'd like one. There are binary search tress that achieve 1, plus very, very tiny thing, arbitrarily tiny, but this is pretty good.

Now, if you don't know Fibonacci numbers, I pull a rabbit out of a hat and I've got this phi to the h. It's kind of magical. There's a much easier way to analyze this recurrence. I'll just tell you because it's good to know but not super critical.

So we have this recurrence, n sub h. This is the computer scientist way to solve the recurrence. We don't care about the constants. This is the theoretical computer scientist way to solve this recurrence. We don't care about constants.

And so we say, aw, this is hard. I've got n sub h minus 1 and n sub h minus 2. So asymmetric. Let's symmetrify. Could I make them both n sub h minus 1. Or could I make them both n sub h minus 2? Suggestions?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Minus 2 is the right way to go because I want to know n sub h is greater than something in order to get a less than down here. By the way, I use that log is monatomic here, but it is, so we're good.

So this is going to be greater than 1 plus 2 times n sub h minus 2. Because if I have a larger height I'm going to have more nodes. That's an easy proof by induction.

So I can combine these into one term. It's simpler. I can get rid of this 1 because that only makes things bigger. So I just have this. OK, now I need a base case, but this looks like 2 the something.

What's the something? H over 2. So I'll just write theta to avoid the base case. 2 to the h over 2. Every two steps of h, I get another factor of 2.

So when you invert and do the log, this means that h is also less than log base 2 of n. Log base 2 because of that. Factor 2 out here because of that factor 2 when you take the log.

And so the real answer is 1.44. This is the correct-- this is the worst case. But it's really easy to prove that it's, at most, 2 log n. So keep this in mind in case we ask you to analyze variance of AVL trees, like in problem set three. This is the easy way to do it and just get some constant times log n. Clear?

All right, so that's AVL trees, why they're balanced. And so if we can achieve this property, that the left and right subtrees have about the same height, we'll be done. So how the heck do we maintain that property? Let's go over here.

Mobius trees are supposed to support a whole bunch of operations, but in particular, insert and delete. I'm just going to worry about insert today. Delete is almost identical. And it's in the code that corresponds to this lecture, so you can take a look at it. Very, very similar.

Let's start with insert. Well, it's pretty straightforward. Our algorithm is as follows. We do the simple BST insertion, which we already saw, which is you walk down the tree to find where that key fits. You search for that key. And wherever it isn't, you insert a node there, insert a new leaf, and add it in.

Now, this will not preserve the AVL property. So the second step is fix the AVL property. And there's a nice concise description of AVL insertion. Of course, how do you do step two is the interesting part. All right, maybe let's start with an example. That could be fun.

Hey, look, here's an example. And to match the notes, I'm going to do insert 23 as a first example. OK, I'm also going to annotate this tree a little bit. So I said we store the heights, but what I care about is which height is larger, the left or the right. In fact, you could just store that, just store whether it's plus 1, minus 1, or 0, the difference between left and right sides.

So I'm going to draw that with a little icon, which is a left arrow, a descending left arrow if this is the bigger side. And this is a right arrow. This is even. Left and right are the same. Here, the left is heavier, or higher, I guess. Here it's even. Here it's left. This is AVL, because it's only one heavier wherever I have an arrow. OK, now I insert 23. 23 belongs-- it's less than 41, greater than 20, less than 29, less than 26. So it belongs here. Here's 23, a brand-new node.

OK, now all the heights change. And it's annoying to draw what the heights are, but I'll do it. This one changes to 1. This is 0. This changes to 2. This changes to 3. This changes to 4.

Anyway, never mind what the heights are. What's bad is, well, this guy's even. This guy's left heavy. This guy's now doubly left heavy. Bad news. OK, let's not worry about above that. Let's just start. The algorithm is going to walk up the tree and say, oh, when do I get something bad?

So now I have 23, 26, 29 in a path. I'd like to fix it. Hmm, how to fix it? I don't think we know how to fix it, so I will tell you how.

Actually, I wasn't here last week. So did we cover rotations?

AUDIENCE: No.

PROFESSOR: OK, good. Then you don't know. Let me tell you about rotations. Super cool.

It's just a tool.

That's x and y. I always get these mixed up. So this is called left rotate of x. OK, so here's the thing we can do with binary search trees. It's like the only thing you need to know. Because you've got search in binary search trees and you've got rotations.

So when I have a tree like this, I've highlighted two nodes, and then there's the children hanging off of them. Some of these might be empty, but they're trees, so we draw them as triangles.

If I just do this, which is like changing which is higher, x or y, and whatever the parent of x was becomes the parent of y. And vice versa, in fact. The parent of y was x, and now the parent of x is y.

OK, the parent of a is still x. The parent of b changes. It used to be y. Now it's x. The parent of c was y. It's still y. So in a constant number of pointer changes, I can go from this to this. This is constant time.

And more importantly, it satisfies the BST order property. If you do an in-order traversal of this, you will get a, x, b, y, c. If I do an in-order traversal over here, I get a, x, b, y, c. So they're the same. So it still has BST ordering.

You can check more formally. b has all the nodes between x and y. Still all the nodes between x and y, and so on. You can check it at home, but this works. We call it a left rotate because the root moves to the left. You can go straight back where you came from. This would be a right rotate of y.

OK, it's a reversible operation. It lets you manipulate the tree. So when we have this picture and we're really sad because this looks like a mess, what we'd like to do is fix it.

This is a path of three nodes. We'd really prefer it to look like this. If we could make that transformation, we'd be happy. And we can. It is a right rotate of 29. So that's what we're going to do.

So let me quickly copy. I want to rotate 29 to the right, which means 29 and 26-- this is x. This is y. I turn them, and so I get 26 here now, and 29 is the new right child. And then whatever was the left child of x becomes the left child of x in the picture. You can check it.

So this used to be the triangle a. And in this case, it's just the node 23. And we are happy. Except I didn't draw the whole tree. Now we're happy because we have an AVL tree again. Good news.

So just check. This is even. This is right heavy. This is even. This is left heavy still. This is left heavy, even, even, even. OK, so now we have an AVL tree and our beauty is restored. I'll do one more example.

Insert 55. We want to insert 55 here. And what changes is now this is even. This is right heavy. This is doubly left heavy. We're super sad. And then we don't look above that until later.

This is more annoying, because you look at this thing, this little path. It's a zigzag path, if you will. If I do a right rotation where this is x and this is y, what I'll get is x, y, and then this is b.

This is what's in between x and y. And so it'll go here. And now it's a zag zig path, which is no better. The height's the same. And we're sad.

I told you, though, that somehow rotations are all we need to do. What can I do? How could I fix this little zigzag? Just need to think about those three nodes, but all I give you are rotations.

AUDIENCE: Perhaps rotate 50.

PROFESSOR: Maybe rotate 50. That seems like a good idea. Let's try it. If you don't mind, I'm just going to write 41, and then there's all the stuff on the left. Now we rotate 50. So 65 remains where it is. And we rotate 50 to the left. So 50 and its child. This is x. This is y. And so I get 55 and I get 50.

Now, this is bad from an AVL perspective. This is still doubly left heavy, this is left heavy, and this is even. But it looks like this case. And so now I can do a right rotation on 65, and I will get-- so let me order the diagrams here.

I do a right rotate on 65, and I will get 41. And to the right I get 55. And to the right I get 65. To the left I get 50. And then I get the left subtree.

And so now this is even, even, even. Wow. How high was left subtree? I think it's still left heavy. Cool. This is what some people call double rotation, but I like to call it two rotations. It's whatever you prefer. It's not really a new operation. It's just doing two rotations.

So that's an example. Let's do the general case. It's no harder. You might say, oh, gosh, why do you do two examples? Well, because they were different. And they're are two cases on the algorithm. You need to know both of them.

OK, so AVL insert. Here we go. Fix AVL property.

I'm just going to call this from the changed node up. So the one thing that's missing from these examples is that you might have to do more than two rotations. What we did was look at the lowest violation of the AVL property and we fixed it. When we do that, there's still may be violations higher up, because when you add a node, you change the height of this subtree, the height of this subtree, the height of this subtree, and the height of this subtree, potentially.

What happened in these cases when I was done, what I did fixed one violation. They were all fixed. But in general, there might be several violations up the tree. So that's what we do. Yeah, I'll leave it at that.

So suppose x is the lowest node that is not AVL. The way we find that node is we start at the node that we changed. We check if that's OK. We update the heights as we go up using our simple rule. And that's actually not our simple rule, but it's erased. We update the height based on the heights of its children. And you keep walking up until you see, oh, the left is twice, two times-- or not two times, but plus 2 larger than the left, or vice versa. Then you say, oh, that's bad. And so we fix it. Yeah, question.

AUDIENCE: So here we continue to [INAUDIBLE].

PROFESSOR: Yes.

AUDIENCE: [INAUDIBLE]. add n to the level [INAUDIBLE] than 1. So add [INAUDIBLE].

PROFESSOR: AVL property's not about levels. It's about left subtrees and right subtrees. So the trouble is that 65-- you have a left subtree, which has height 2-- or sorry, height 1, I guess-- because the longest path from here to a leaf is 1.

The right subtree has height negative 1 because it doesn't exist. So it's one versus negative 1. So that's why there's a double arrow. Yeah, good to ask. It's weird with the negative 1s. That's also why I wanted to define those negative 1s to be there, so the AVL property is easier to state. Other questions?

All right. Good. I think I want a symmetry assumption here. I don't know why I wrote right of x. I guess in modern days we write x dot right. Same thing. OK, I'm going to assume that the right child is the heavier one like we did before. Could be the left. It's symmetric. It doesn't matter.

So now there are two cases, like I said.

I'm going to use this term right heavy because it's super convenient. OK, right heavy is what I've been drawing by a descending right arrow. Balance is what I've been drawing by a horizontal line. OK, so we're just distinguishing between these two cases. This turns out to be the easy case.

So we have x, y, a, b, c. Why are we looking at the right child? Because we assumed that the right one is higher, so that x was right heavy. So this subtree as I've drawn it is higher than the left one by 2, in fact. And what we do in this case is right rotate of x. And so we get x, y, a, b, c. I could have drawn this no matter what case we're in, so we need to check this actually works. That's the interesting part. And that's over here.

OK, so I said x is right heavy, in fact doubly so. y is either right heavy or balanced. Let's start with right heavy. So when we do this rotation, what happens to the heights? Well, it's hard to tell. It's a lot easier to think about what the actual heights are than just these arrows.

So let's suppose x has height k. That's pretty generic. And it's right heavy, so that means the y has height k minus 1. And then this is right heavy, so this has height k minus 2. And this is something smaller then k minus 2.

In fact, because this is AVL, we assume that x was the lowest that is not AVL. So y is AVL. And so this is going to be k minus 3, and this is going to be k minus 3 because these differ by 2. You can prove by a simple induction you never get more than 2 out of whack because we're just adding 1, off by 1. So we got off by 2.

So this is the bad situation. Now we can just update the heights over here. So k minus 3 for a, k minus 3 for b, k minus 2 for c. Those don't change because we didn't touch those trees, and height is about going down, not up.

And so this becomes k minus 2, and this becomes k minus 1. And so we changed the height of the root, but now you can see that life is good. This is now balanced between k minus 3 and k minus 3. This is now balanced between k minus 2 and k minus 2.

And now the parent of y may be messed up, and that's why after this we go to the parent of y, see if it's messed up, but keep working our way up. But it worked. And in the interest of time, I will not check the case where y is balanced, but it works out, too.

And see the notes. So the other case is where we do two rotations. And in general, so here x was doubly right heavy. And the else case is when the right child of x, which I'm going to call z here, is left heavy. That's the one remaining situation.

You do the same thing, and you check that right rotating and left rotating, which makes the nice picture, which is x, y, z, actually balances everything and you restore the AVL property. So again, check the notes on that.

I have a couple minutes left, and instead I'd like to tell you a little bit about how this fits into big-picture land. Two things I want to talk about. One is you could use this, of course, to sort, which is, if you want to sort n numbers, you insert them and you do in-order traversal.

How long does this take? In-order traversal takes linear time. That's the sense in which we're storing things in sorted order. Inserting n items-- well, each insert takes h time, but now we're guaranteed that h is order log n. So all the insertions take log n time each, n log n total. So this is yet another way to sort n items in n log n time, in some ways the most powerful way.

We've seen heaps, and we've seen merge sort. They all sort. Heaps let you do two operations, insert and delete min, which a lot of times is all you care about, like in p set two. But these guys, AVL trees, let you do insert, delete, and delete min. So they're the same in those senses, but we have the new operation, which is that we can do find next larger and next smaller, aka successor and predecessor.

So you can think about what we call an abstract data type. These are the operations that you support, or that you're supposed to support. If you're into Java, you call this an interface. But this is an algorithmic specification of what your data structure is supposed to do.

So we have operations like insert and delete. We have operations like find the min and things like successor and predecessor, or next larger, next smaller. You can take any subset of these and it's an abstract data type. Insert, delete, and min is called a priority queue.

So if you just take these first two, it's called a priority queue. And there are many priority queues. This is a generic thing that you might want to do. And then the data structure on the other side is how you actually do it. This is the analog of the algorithm.

OK, this is the specification. You want a priority queue. One way to do it is a heap. Another way to do it is an AVL tree. You could do it with a sorted array. You could do lots of sub-optimal things, too, but in particular, heaps get these two operations. If you want all three, you basically need a balanced binary search tree.

There are probably a dozen balanced binary search trees out there, at least a dozen balanced search trees, not all binary. They all achieve log n. So it doesn't really matter. There are various practical issues, constant factors, things like that.

The main reason you prefer a heap is that it's in place. It doesn't use any extra space. Here, you've got pointers all over the place. You lose a constant factor in space. But from a theoretical standpoint, if you don't care about constant factors, AVL trees are really good because they get everything that we've seen so far and log n. And I'll stop there.